Upgrading Fortran Source Code Using Automatic Refactoring

Dominic Orchard Andrew Rice
Computer Laboratory, University of Cambridge
dominic.orchard@cl.cam.ac.uk andrew.rice@cl.cam.ac.uk

Abstract

Many of the computer models used in scientific research have been developed in Fortran over many years. This evolutionary process means these models often use deprecated language features and idioms that impede software maintenance, understandability, extension, and verification. To mitigate this, we built *CamFort*, an open-source automatic refactoring tool for upgrading Fortran source code. We describe functionality in CamFort for removing *equivalence statements* and *common blocks*, and for introducing structured data types, and give examples of how these transformations can benefit codebase robustness.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: FORTRAN, Fortran 90; K.6.3 [*Software Management*]: Software maintenance

Keywords Fortran, refactoring, language evolution, computational science, Haskell

1. Introduction

The approach of *computational science* uses computer technology to collect and analyse data, and to express complex scientific theories completely via computer models. Such models share the software quality goals of traditional software engineering, such as verifiability, maintainability, understandability, and portability. However, many of these goals are not readily achieved in the sciences. For example, the need for better verification has been highlighted by several recent cases, including researchers in biology retracting five papers (three from *Science*) due to a single programming error invalidating their results [4]. The importance of model maintainability and sustainability is also being increasingly recognised (*e.g.*, the EPSRC released £5 million in 2010 to support long-term software sustainability in science [2]).

WRT '13, October 27, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2604-9/13/10. . . $15.00.
http://dx.doi.org/10.1145/2541348.2541356

In computational science, imperative languages such as Fortran and C remain popular, often due to the legacy of preceding models. With over fifty years of history, Fortran has shown remarkable longevity, in part due to its evolution with frequent (re)standardization efforts [3], and the continued development of tools and compilers. Fortran's retirement has been advocated for over 20 years [1], yet it remains the standard in many research labs around the world. Other than legacy, Fortran is attractive to scientists because of high-level array support, low runtime overhead (hence predictable and controllable performance), and ease of optimisation.

Due to Fortran's long history, many compilers continue to support the older language standards of the 1960s and 70s. This perpetuates the existence of legacy code using deprecated features that are now widely recognised to be dangerous. Such features are a source of program errors and prevent further transformations. We therefore advocate, as have others [8, 11], for automatic refactoring tools to provide a pathway to *upgrading*, or modernising, legacy code.

To this end, we built *CamFort*, an analysis and automatic refactoring tool to support maintenance, verification, understanding, and further refactoring of Fortran code. Our tool currently focuses on language features and programming idioms related to manual memory and data management. We show here three refactorings from CamFort: *equivalence statement elimination* (Section 2.1), *common block elimination* (Section 2.2), and *derived data type introduction* to replace a manual data-structuring idiom (Section 2.3).

Context of our work We are running an inter-disciplinary project at the University of Cambridge between computer scientists and natural scientists to leverage state-of-the-art programming language research for more effective programming in the sciences. We are studying a number of models, mostly written in Fortran, developed within the university. CamFort if part of our Fortran research infrastructure, providing the basis of additional tools for collecting data on programming idioms (to inform future language and tool designs) and for experimenting with new language extensions.

CamFort is open source and available online at `http://www.cl.cam.ac.uk/research/dtg/camfort`

Related work There are number of restructuring and refactoring tools for Fortran. A recent survey is provided by Tinetti and Méndez [11]. Our refactorings are not currently

supported by other tools, with the exception of refactoring common blocks to modules (Sect. 2.2) in Photran [9], plusFORT [10], and VAST/77to90 [13]. Photran provides various Fortran refactorings via an Eclipse IDE extension. It can be adapted for more specific tasks, for example refactoring global variables to allow MPI-based parallel programs to be converted to Adaptive MPI implementations [7].

Implementation CamFort is essentially a compiler front end, parsing Fortran files to an abstract syntax tree (AST) representation, performing standard and Fortran-specific program analyses, and AST transformations. CamFort outputs analysis reports and Fortran code. Much of Fortran is recognised by the parser, from Fortran 66 to 2003 standards.

CamFort is implemented as an embedded domain-specific language in Haskell for rapid prototyping of program analyses and transformations, in a compositional style. Standard components include type analysis, dataflow analyses, and inspection and transformation of nodes in a subtree of a certain kind (*e.g.*, all assignments). Refactorings may produce redundant code which is optimised separately by dead-code elimination to reduce the footprint of refactored code.

Data-type generic programming approaches are used for rapid development and a relatively small implementation. For example, a pleasingly terse algorithm merges (possibly refactored) ASTs with their source codebase, producing a refactored codebase with textual changes only at refactored nodes, *i.e.*, preserving formatting and comments of the original codebase. Since the algorithm is data-type generic, it does not need rewriting if the AST definition changes.

2. Refactoring with CamFort

Throughout, the term *program unit* refers to programs, subprograms (functions or subroutines), or modules.

2.1 Equivalence statement elimination

Equivalence statements declare a number of variables to be aliased to the same memory location, possibly of different types, with syntax of the form:

```
equivalence (var1, var2 [, vars])
```

where `[, vars]` is an optional comma-separated list of variables; an equivalence statement has two or more variables (which may also be arrays or constant offsets into arrays).

An equivalence statement that aliases variables of different types introduces implicit *unsafe casting* such that a piece of memory is viewed through the lens of different types.

Equivalence statements are widely recognised as dangerous (they are banned in the UK government's Met Office Fortran coding standard [5]). In the past, equivalence statements were used to reduce memory usage. However, memory capacity is now much less of a concern. Instead, any aliasing in a program is an opportunity for unintended side effects, for example when equivalenced variables are *live* in the same region of code. Furthermore, if the types differ, then the implicit unsafe type casts are a potential source of data errors, which are hard to understand statically.

```
1 integer :: x, y, i
2 equivalence (x, y)
3 x = 0
4 x = 5
5 do i = 1, x
6    x = x + i
7 end do
8 print '(i8, i8)', x, y
```

(a) Example with equivalence

```
1 integer :: x, y, i
2
3 x = 0
4 x = 5
5 do i = 1, x
6    x = x + i
7    y = x
8 end do
9 print '(i8, i8)', x, y
```

(b) CamFort refactoring of (a)

```
(2,0): removed equivalence
(3,0): added copy: y = x due to refactored equivalence
(4,0): added copy: y = x due to refactored equivalence
(6,3): added copy: y = x due to refactored equivalence
r(3,0),(4,0): removed dead code from refactoring
r(7,0): WARNING: y and x interfere here (possible bug)
```

(c) CamFort refactoring log

Figure 1. Example refactoring of equivalence statements

Refactoring Our refactoring eliminates equivalence statements, inserting *copy statements*, *e.g.*, $\texttt{var}_1$ `=` $\texttt{var}_2$, whenever $\texttt{var}_2$ is updated and $\texttt{var}_1$ and $\texttt{var}_2$ are equivalenced. If the types of equivalenced variables differ, an explicit unsafe type cast is inserted in the copy statement, provided by Fortran 90's `transfer` intrinsic. For example, $\texttt{var}_1$ `=` `transfer(`$\texttt{var}_2$`,`$\texttt{var}_1$`)` copies $\texttt{var}_2$ to $\texttt{var}_1$ at the type of $\texttt{var}_1$, preserving the bit-level data.

Figure 1 shows an example program fragment and its refactoring, along with the refactoring log provided to the user. This information could inform an IDE's feedback to the user. Note, three copy statements are generated, but two are dead-code eliminated. The statement `x = 0` is dead in the original program, but is not removed in order to preserve the original program structure; dead-code elimination is only performed on code introduced by the refactoring.

Benefits Our refactoring makes explicit the implicit aliasing and casting introduced by equivalence statements. This serves to highlight potential sources of error. For example, the refactoring makes explicit when equivalenced variables are live at the same program point by introducing a copy statement that is not subsequently dead-code eliminated, *e.g.*, Figure 1(b) line 7; both `x` and `y` are live at line 7 hence the inserted copy is not eliminated. This implies interference between the two variables, which may be unintended and therefore a source of program error. This is made explicit in the code and reported to the user.

Equivalence elimination also assists further refactoring. For example, equivalence statements impede the refactoring of a variable's type (*e.g.*, from single- to double-precision floating point, a common change in scientific code). If a variable is equivalenced, changing its type introduces a potential source of data error through unsafe casting. Eliminating the equivalence statements makes the casts explicit, which could

be replaced with safe data conversions, or all equivalenced variables could be refactored to the same type.

2.2 Common block elimination

Common blocks provide sharing between program units, akin to global variables. Their declaration comprises a list of variables (declared in the containing scope), with syntax:

```
common /name/ vars
```

where `vars` is a list of comma-separated variables (the fields of the block) and `/name/` is an optional name. There may be arbitrarily many named common blocks in a program and a single unnamed common block (declared by eliding `/name/`). Each program unit which shares values via a particular common block includes a common block declaration of that name, but may declare fields of different name, type, and even length. Figure 2(a) gives an example of sharing between a main block and subroutine via an unnamed common block, with differing variable names (*cf.* line 3 and 9).

Common blocks were used extensively in older, pre-Fortran 90 code for sharing values and saving memory [11]. However, they are widely known to be problematic [5]. Similarly to equivalence statements, they allow data aliasing which may lead to unexpected interference, as well as implicit unsafe casts. Further, common blocks are not checked for consistency between program units, which may be separately compiled with differing common block declarations.

CamFort provides two ways to refactor common blocks: 1) explicit parameter passing, and 2) *modules*.

Refactoring to parameter passing Variables shared by a common block are refactored as parameters, illustrated in Figure 2(b). Since Fortran uses call-by-reference (or call-by-copy-restore) any updates to `a` and `b` in `bar` are seen in the main program. This refactoring is useful when common blocks are used minimally in subroutines/functions and the inserted parameters do not clutter the code. Explicit unsafe casts can also be inserted (using `transfer`) if required.

Refactoring to modules A more elegant refactoring uses *modules* (discussed previously [8]) replacing common blocks with a single declaration point for the variables shared between program units, unifying the names and types [6]. Figure 2(c) illustrates the approach. Where variable names differ, these variables are matched to the module declaration using the renaming syntax of `use` statements (*e.g.*, line 7).

Benefits Similarly to equivalence elimination, common block elimination exposes potential data errors from implicit casts, potential sources of interference due to aliasing, and facilitates further refactorings (*e.g.*, type refactorings).

Our refactoring to modules applies only to common blocks with the same field lengths and types, since modules unifies these. This is more powerful than the approach of plusFORT, where common blocks must also use the same variable names [10][§2.7.6]. Our parameter-passing refactoring however allows different field lengths and types, and

```
1  program foo
2    integer :: x, y
3    common x, y
4    call bar()
5  end program foo
6
7  subroutine bar()
8    integer :: a, b
9    common a, b
10   a = a + b
11 end subroutine bar
```

(a) Common block example

```
1  program foo
2    integer :: x, y
3
4    call bar(x, y)
5  end program foo
6
7  subroutine bar(a, b)
8    integer :: a, b
9
10   a = a + b
11 end subroutine bar
```

(b) Parameter refactoring

```
1  program foo
2    use common
3    call bar()
4  end program foo
5
6  subroutine bar()
7    use common, a => x, b => y
8    a = a + b
9  end subroutine bar
```

`common.f90`:

```
1  module common
2    integer :: x
3    integer :: y
4  end module common
```

(c) CamFort refactoring of (a) using modules

Figure 2. Example refactoring of a common block

is therefore more general. However, it introduces excessive code clutter if common blocks are used frequently. In contrast, the module approach gives a single, clear definition of the shared data, scaling better with more complex codebase.

2.3 Derived data type introduction

In the past, a common idiom used arrays to group large parameter sets, manually implementing a kind of *record type*. In this idiom, an array provides manual records with named fields defined by variables bound to array indices, manually defining record projection. For example, Figure 3(a) declares an array `p` of parameters of distance and time values, where variables `d1`, `d2`, `t1`, `t2` provide projections, *e.g.*, `p(d1)`. These manual projections require programmer effort and are a potential source of error if projections overlap unintentionally. Further, we have seen cases where these arrays are large and unwieldy yet could be decomposed into more meaningful subsets based on projection variable usage in the program: some subsets of projection variables are used disjointly or with small amounts of overlapping use.

Refactoring CamFort refactors this idiom into Fortran 90 *derived data types* (essentially *named records* resembling C structs [6]). Figure 3(b) shows example syntax.

Manual records are decomposed based on semantic relationships between the projection variables. The refactoring first constructs an *interference graph* (as is used in register colouring algorithms) for variables which index an array (*e.g.*, `d1` in `p(d1)`), where two variables have an edge between if a value is computed involving both variables.

A derived data type is generated from each disconnected subgraph of the interference graph, where each node of the subgraph generates a variable declaration of the type of the original array elements. Data type names are generated

```
real p(0:3)
integer :: d1, d2, t1, t2

d1 = 0
t1 = 1
d2 = 2
t2 = 3

p  = ... ! init p
v1 = p(d1) / p(t1)
v2 = p(d2) / p(t2)
```

(a) Manual records example

```
type X1
  real :: d1, t1
end type X1
type(X1) p1

type X2
  real :: d2, t2
end type X2
type(X2) p2

p  = ... ! init p
v1 = p1%d1 / p1%t1
v2 = p2%d2 / p2%t2
```

(b) CamFort refactoring of (a)

Figure 3. Example refactoring of manual records

by taking the per-character mode (most frequent character) from the variable names within, with a wildcard character `X` if there is no unique mode. Each derived data type declaration is followed by a value declaration of this type.

Finally, the original declarations and value definitions of the projection variables are removed from the program and array indexing is replaced with derived data type projection, *e.g.*, `p%d1` replaces `p(d1)`.

Figure 3(b) shows the refactoring for our example, where `d1`, `t1` and `d2`, `t2` form disconnected subgraphs in the interference graph. Note, the refactoring only succeeds if projection variables are non-overlapping static constants (up to *copy propagation*). Otherwise, the refactoring reports which constants overlap or cannot be statically determined.

The refactoring allows further decomposition of derived data types by taking *minimal cuts* of subgraphs of the interference graph (*i.e.*, splitting subgraphs into two by removing the smallest number of edges). This process can be iterated. Further work is to refactor data types by merging adjacent declarations that have the same structure (types), thus increasing abstraction and reducing code size. For example, Figure 3(b) would have its two data types merged into one.

Benefits In the manual approach, extending the record means increasing the array size, adding a new lookup variable, and propagating these changes throughout. Derived data type introduction therefore makes code much easier to maintain, understand, and extend. The refactoring also identifies errors in manual record projections arising from unintended overlap of projection variables.

3. Conclusion & further work

We advocate for refactoring of legacy code as languages evolve to avoid "sedimentary programs" with layers of past and present code. CamFort goes toward upgrading Fortran code to a modern form, eliminating deprecated features and introducing structure data types. This assists the maintenance, verification, extension, and understandability of code.

The Haskell embedded domain-specific language approach makes CamFort well-placed to support further refactorings (*e.g.*, those suggested by Overbey and Johnson [8]). There is scope for improving this approach further (*e.g.*, adapting *source-code query languages* [12]) to ease the development of new analyses and refactorings. We also intend to use the CamFort analysis framework as a basis for collecting data on programming patterns/idioms used in scientific modelling. For example, how common are parallelisable loops that could be executed on a GPU? This will provide invaluable data for future language and tool designs.

Acknowledgments Grateful thanks to Andy Hopper for his support, Sam Aaron, Andrew Friend, Alan Mycroft, and Raoul-Gabriel Urma for their comments, and Mark Sydall for discussing Fortran transformation. This research was supported by a Google Focused Research Award.

References

[1] D. Cann. Retire Fortran? A debate rekindled. In *Proc. ACM/IEEE Conf. on Supercomputing*, pages 264–272, 1991.

[2] EPSRC. Core e-Science program strategy, Retr. Aug 2013. `http://www.epsrc.ac.uk/research/ourportfolio/themes/researchinfrastructure/subthemes/einfrastructure/escience/Pages/strategy.aspx`.

[3] M. N. Greenfield. History of FORTRAN standardization. In *Proceedings of the National Computer Conference*, pages 817–824. ACM, 1982.

[4] Z. Merali. Computational science: Error, why scientific programming does not compute. *Nature*, 467(7317):775–777, 2010.

[5] Met office: Fortran 90 standards, Retrieved Aug 2013. `http://research.metoffice.gov.uk/research/nwp/numerical/fortran90/f90_standards.html`.

[6] M. Metcalf. Why Fortran 90? Technical report, CM-P00065587, 1991.

[7] S. Negara, G. Zheng, K.-C. Pan, N. Negara, R. E. Johnson, L. V. Kalé, and P. M. Ricker. Automatic MPI to AMPI program transformation using Photran. In *Euro-Par 2010 Parallel Processing Workshops*, pages 531–539. Springer, 2011.

[8] J. L. Overbey and R. E. Johnson. Regrowing a language: refactoring tools allow programming languages to evolve. In *ACM SIGPLAN Notices*, volume 44, pages 493–502, 2009.

[9] Photran – An Integrated Development Environment and Refactoring Tool for Fortran, Retrieved July 2013. `http://www.eclipse.org/photran/`.

[10] PlusFORT Manual, Retrieved October 2013. `http://www.polyhedron.com/plusfortmanual/index.html`.

[11] F. G. Tinetti and M. Méndez. Fortran legacy software: source code update and possible parallelisation issues. In *ACM SIGPLAN Fortran Forum*, volume 31, pages 5–22. ACM, 2012.

[12] R.-G. Urma and A. Mycroft. Source-code queries with graph databases–with application to programming language usage and evolution. *Science of Computer Programming*, 2013.

[13] VAST/77to90, Retrieved October 2013. `http://www.crescentbaysoftware.com/vast_77to90.html`.